Porting and Optimizing Multimedia Codecs for AMD64 architecture on Microsoft® Windows®

July 21, 2004

Abstract
This paper provides information about the significant optimization techniques used for multimedia encoders and decoders on AMD64 platforms running 64-bit Microsoft® Windows®. It walks the developers through the challenges encountered while porting codec software from 32 bits to 64 bits. It also provides insight into the tools and methodologies used to evaluate the potential benefit from these techniques. This paper should serve as a practical guide for the software developers seeking to take codec performance on the AMD Athlon™ 64 and AMD Opteron™ family of processors to its fullest potential.

Contents

Windows Hardware Engineering Conference
Author's Disclaimer and Copyright:

Introduction

The 64-bit computing wave accompanied by its challenges and benefits is here. It is impossible for developers seeking to make their applications perform faster and more efficient to ignore this wave.

All existing 32-bit applications can be run on 64-bit Microsoft® Windows® on AMD64 processors without any detriment to performance. But, there is an opportunity to leverage even higher performance by porting to 64 bits. The inherent compute intensive nature of multimedia encoders and decoders, commonly known as codecs, makes these applications ideal candidates for porting to 64 bits.

This paper will describe the various challenges that developers will encounter while porting codecs to AMD64 platforms. It will explain the rationale behind various porting and optimization techniques and demonstrate them with examples from codecs. The paper will elaborate on how the CodeAnalyst tool is used to profile codec software. It will briefly talk about some development tools that can be used to facilitate the porting.

This paper assumes that the reader is familiar with MMX™ and SSE/SSE2 instruction sets and various aspects of codec technology.

Porting Challenges

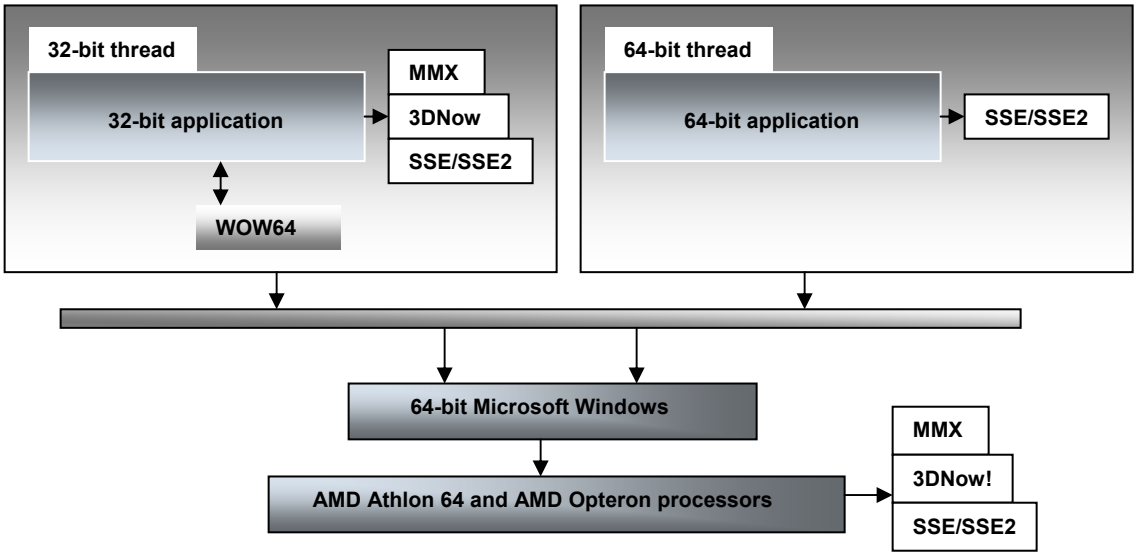
Before discussing the optimization details, it is important for the developer to understand some of the initial obstacles to porting codecs from 32 bits to 64 bits and how to circumvent them. These techniques also expose some of the underlying portability and performance benefits that can improve the maintainability and quality of codecs.

Avoid MMX™ And 3DNow! ™ Instructions In Favor Of SSE/SSE2 Instructions

64-bit Microsoft Windows on AMD64 platforms can run both 32-bit and 64-bit applications side by side seamlessly. 32-bit applications are run in a 32-bit legacy mode called WOW64 (Windows on Windows64). 64-bit applications are run in the 64-bit native mode.

AMD Athlon™ 64 and AMD Opteron™ processors have support for all x86 instruction extensions; MMX, SSE, SSE2 and 3DNow!. 64-bit Microsoft Windows supports MMX, SSE, SSE2 and 3Dnow! for 32-bit applications under WOW64. However, 64-bit Microsoft Windows does not strongly support MMX and 3Dnow! instruction sets in the 64-bit native mode. The preferred instructions sets are SSE and SSE2.

Figure 1: Instruction set architecture support in 64-bit Microsoft Windows for AMD64 platforms.



One of the biggest challenges for porting codecs to AMD64 platforms lies in the fact that several components of audio and video codecs have been optimized with SIMD assembly modules that use the MMX instruction set. Such codec modules can be run without any modification or recompilation as 32-bit applications on WOW64. However, such modules will have to be rewritten to use the SSE/SSE2 instructions.

The advantage of using the SSE/SSE2 instruction set lies in the fact that it operates on 128 bit registers compared to the 64-bit registers used by MMX instructions. Since SSE/SSE2 instructions can operate on data twice the size of MMX instructions, fewer instructions are required. This improves code density and allows

for better instruction cache and decoding performance. 64-bit native mode has access to sixteen XMM registers. The 32-bit legacy mode only has eight XMM registers.

Use Intrinsics Instead Of Inline Assembly

Several performance critical components of audio and video codecs are usually written as inline assembly modules. Using inline assembly suffers from the inherent burden of working at the low level. The developer would also have to understand the register allocation surrounding the inline assembly to avoid performance degradation. This can be a difficult and error prone task.

The 64-bit Microsoft Windows C/C++ compiler does not support inline assembly in the native 64-bit mode. It is possible to separate the inline assembly modules into stand-alone callable assembly functions, port them to 64 bits, compile them with the Microsoft assembler and link them into the 64-bit application.

However, this is a cumbersome process. These assembly functions will suffer from additional calling linkage overhead. The assembly developer would have to be concerned with the application binary interface specified for 64-bit Microsoft Windows on AMD64 architecture. Bulky prologues and epilogues, used to save and restore registers, could lead to performance degradation.

The solution offered by the Microsoft Windows compilers is a C/C++ language interface to SSE/SSE2 assembly instructions called *intrinsics*. Intrinsics are inherently more efficient than called functions because no calling linkage is required and the compiler will inline them. Furthermore, the compiler will manage things that the developer would usually have to be concerned with, such as register allocation.

Implementing with intrinsics makes the code portable between 32 and 64 bits. This provides for ease of development and maintenance. Using intrinsics allows the 64-bit Microsoft compiler to use efficient register allocation, optimal scheduling, scaled index addressing modes and other optimizations that are tuned for the AMD64 platform.

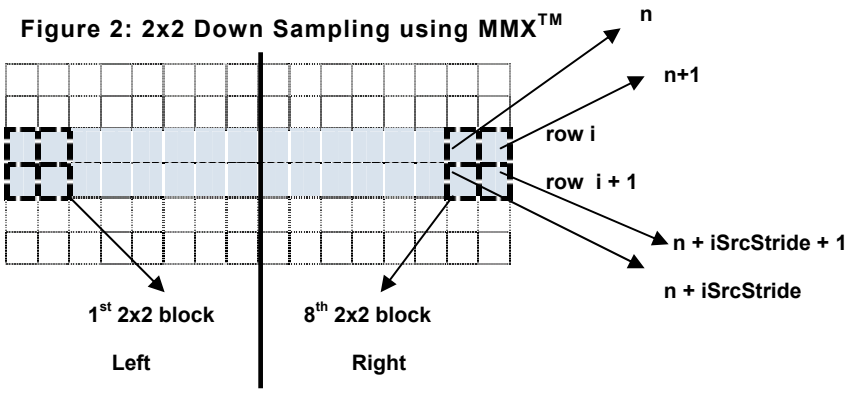
Example:

Let us look at a codec example where MMX assembly can be converted to intrinsics.

Without getting too engrossed in the theory of downsampling, let’s go through an algorithm for 2x2 downsampling, where a pixel in the scaled output image is an average of pixels in a corresponding 2x2 block in the input image. Pixels n, n + 1 (immediate right pixel), n + iSrcStride (immediate below pixel), n + iSrcStride + 1 (immediate diagonal right pixel) are averaged in the input image to generate a pixel in the output image. Here iSrcStride refers to the width of the block.

This is repeated for every 2x2 block in a 16x16 block of pixels. Let us consider two rows, i and i +1, of a 16x16 block being processed at a point in time. These two rows in a 16x16 block will contain eight 2x2 blocks.

Figure 2: 2x2 Down Sampling using MMX™



The MMX SIMD algorithm, shown below, processes 2 rows of a 16x16 block per iteration of the loop. The loop first processes the left side of the row or four 2x2 blocks and then processes the right side of the row or the next four 2x2 blocks. The loop has to iterate 8 times to process the entire 16x16 block.

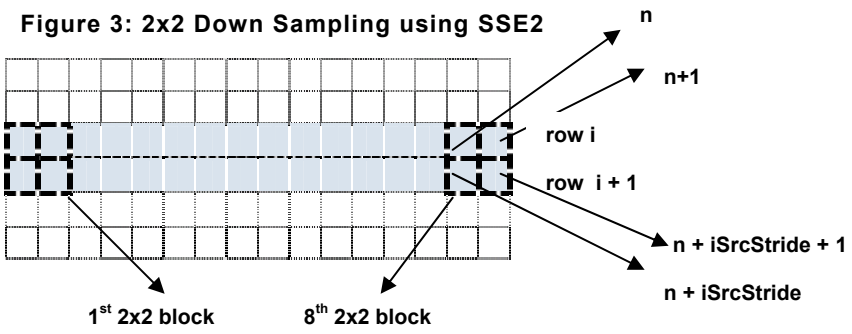
Listing 1: 2x2 Down Sampling with MMX instructions

```
;pSrc = 16 byte aligned pointer to a chunk of 1 byte pixels
;pDst = 16 byte aligned pointer to a chunk of 1 byte pixels
mov    eax, 8
mov    esi, pSrc
mov    edi, pDst
mov    ecx, iSrcStride
mov    edx, iDstStride
mov    ebx, 0x00FF00FF
movd   mm4, ebx
punpcklwd   mm4, mm4
Loop:
       movq        mm7, [esi]            //Load row i of left four
2x2 pixels
       movq        mm5, [esi + ecx]      //Load row i+1 of left four
2x2 pixels

       movq        mm6, mm7
       psrlw       mm6, 8
       pavgb       mm7, mm6
       movq        mm2, mm5
       psrlw       mm2, 8
       pavgb       mm5, mm2

       pavgb       mm7, mm5
       pand        mm7, mm4
       packuswb    mm7, mm7
       movd        [edi], mm7

       add         esi, 8                //Point to right four 2x2
pixels
       add         edi, 4

       movq        mm7, [esi]            //Load row i of right four
2x2 pixels
       movq        mm5, [esi + ecx]      //Load row i+1 of right four
2x2 pixels

       movq        mm6, mm7
       psrlw       mm6, 8
       pavgb       mm7, mm6
       movq        mm2, mm5
       psrlw       mm2, 8
```

```
        pavgb       mm5, mm2

        pavgb       mm7, mm5
        pand        mm7, mm4
        packuswb    mm7, mm7
        movd        [edi], mm7

        add         esi, ecx;             //Advance pSrc by 2 rows
since a 2x2 block is processed each time
        add         esi, ecx
        add         edi, edx
        dec         eax
        jnz         Loop
```

SSE/SSE2 instructions operate on 16-byte XMM registers instead of the 8-byte MMX registers used by MMX instructions. The same algorithm implemented in SSE2 would be able to process the same number of pixels in half the number of instructions per iteration.

Figure 3: 2x2 Down Sampling using SSE2



Left and Right sides processed simultaneously

Listing 2: 2x2 Down Sampling with SSE2 Instructions

```
;pSrc = 16 byte aligned pointer to a chunk of 1 byte pixels
;pDst = 16 byte aligned pointer to a chunk of 1 byte pixels
mov     eax, 8
mov     esi, pSrc
mov     edi, pDst
mov     ecx, iSrcStride
mov     [ebx], 0x00FF00FF
movd    xmm4, [ebx]
punpcklwd    xmm4, xmm4

Loop:
        movdqa      xmm7, [esi]           //Load row i of all eight
2x2 pixels
        movdqa      xmm5, [esi + ecx]     //Load row i+1 of all eight
2x2 pixels

        movdqa      xmm6, xmm7
        psrlw       xmm6, 8
        pavgb       xmm7, xmm6
        movdqa      xmm2, xmm5
        psrlw       xmm2, 8
        pavgb       xmm5, xmm2

        pavgb       xmm7, xmm5
        pand        xmm7, xmm4
        packuswb    xmm7, xmm7
        movdqa      [edi], xmm7
```

```
       add         esi, ecx             //Advance pSrc by 2 rows
since a 2x2 block is processes each time
       add         esi, ecx
       add         edi, edx
       dec         eax
       jnz         Loop
```

Since the AMD Athlon 64 and AMD Opteron processors implement the general purpose and MMX register files separately, it is expensive to move data between the two sets of registers. Likewise, it is also expensive to move data between general purpose and XMM registers used by SSE/SSE2.

Note that the MMX algorithm loads integer constant 0x00FF00FF data into a general-purpose register and moves the general-purpose register into an MMX register. Such register moves should be avoided and data should be loaded into MMX or XMM registers from memory where possible. This has been fixed in the SSE2 sequence above.

When using Intrinsics, the Microsoft Windows compiler for AMD64 platforms will take care of this detail and the user does not have to worry about it. The compiler will also perform the most optimal register allocation in this case. The alternate intrinsic implementation that generates the SSE2 assembly is shown below:

Listing 3: 2x2 Down Sampling with SSE2 Intrinsics

```
;pSrc = 16 byte aligned pointer to a chunk of 1 byte pixels
;pDst = 16 byte aligned pointer to a chunk of 1 byte pixels
__m128i Cur_Row, Next_Row, Temp_Row;
int w = 0x00FF00FF;
__m128i Mask= _mm_set1_epi32 (w);
for (i=0; i<8; i++)
{
       Cur_Row      = _mm_load_si128((__m128i *)pSrc);
       //Load row i of all eight 2x2 pixels
       Next_Row     = _mm_load_si128((__m128i *)(pSrc +
iSrcStride));
       //Load row i+1 of all eight 2x2 pixels


       Temp_Row     = Cur_Row;
       Temp_Row     = _mm_srli_epi16 (Temp_Row, 8);
       Cur_Row      = _mm_avg_epu8 (Cur_Row, Temp_Row);

       Temp_Row     = Next_Row;
       Temp_Row     = _mm_srli_epi16 (Temp_Row, 8);
       Next_Row     = _mm_avg_epu8 (Next_Row, Temp_Row);

       Cur_Row      = _mm_avg_epu8 (Cur_Row, Next_Row);
       Cur_Row      = _mm_and_si128 (Cur_Row, Mask);
       Cur_Row      = _mm_packus_epi16 (Cur_Row, Cur_Row);

       mm_store_si128((__m128i *)(pDst), Cur_Row);
       pSrc+=iSrcStride;
       pSrc+=iSrcStride;
       //Advance pSrc by 2 rows since a 2x2 block is processes each
time
       pDst+=iDstStride;
}
```

The 64-bit Microsoft Windows C/C++ compiler exposes `__m128`, `__m128d` and `__m128i` data types for use with SSE/SSE2 Intrinsics. These data types denote 16-byte aligned chunks of packed single, double and integer data respectively.

The following correspondence between intrinsics and SSE2 instructions can be drawn from the above example. For a complete listing of intrinsics and further details, the user should refer to the Microsoft MSDN Library.

SSE2 Intrinsics	SSE2 Instructions
_mm_load_si128	movdqa
_mm_srli_epi16	psrlw
_mm_avg_epu8	pavgb
_mm_and_si128	pand
_mm_packus_epi16	packuswb

Use Portable Scalable Data Types

Codec developers should use portable and scalable data types for pointer manipulation. While `int` and `long` data types remain 32 bits on 64-bit Microsoft Windows running on AMD64 platforms, all pointers expand to 64 bits. Using these data types for type casting pointers will cause 32-bit pointer truncation.

The recommended solution is the use of `size_t` or other polymorphic datatypes like `INT_PTR`, `UNIT_PTR`, `LONG_PTR` and `ULONG_PTR` when dealing with pointers. These data types, exposed by the 64-bit Microsoft C/C++ compilers, automatically scale to match the 32/64 modes.

Example:

Let us look at an example of how a code sequence can be changed to avoid pointer truncation.

```
char pSrc[64];
char* pSrc_ptr = (char*)
       ((Int)pSrc+15)&~15);
```

→

```
char pSrc[64];
char* pSrc_ptr = (char*)
       (((INT_PTR)pSrc+15)&~15);
```

While dealing with structures containing pointers, users should be careful not to use explicit hard coded values for the size of the pointers. Let us look at an example of how this can be erroneous.

```
typedef struct _clock_object {
    int       id;
    int       size;
    int       *pMetaDataObject;
} clock_object;
```

```
void Init_clock(clock_object
*object){
object->id = dummy;
object->size = sizeof(object)-
4;
…}
```

→

```
void Init_clock(clock_object
*object){
object->id = dummy;
object->size = sizeof(object)-
sizeof(pMetaDataObject);
…}
```

Optimization Techniques

There are several optimization techniques that the developer can use to get high performance on the 64-bit Microsoft Windows for AMD64 platforms. This paper will illustrate some of them with examples from various codecs. For a more extensive documentation on optimizations, the user should refer to the Software Optimization guide for AMD Athlon 64 and AMD Opteron processors. Other resources are also available at the AMD64 developer resource kit site.

Extended 64-Bit General Purpose Registers

One of the biggest advantages in the 64-bit native mode over the 32-bit legacy mode is the extended width of general-purpose registers. While using 64-bit registers lengthens the instruction encoding, it can significantly reduce the number of instructions. This in turn improves the code density and throughput.

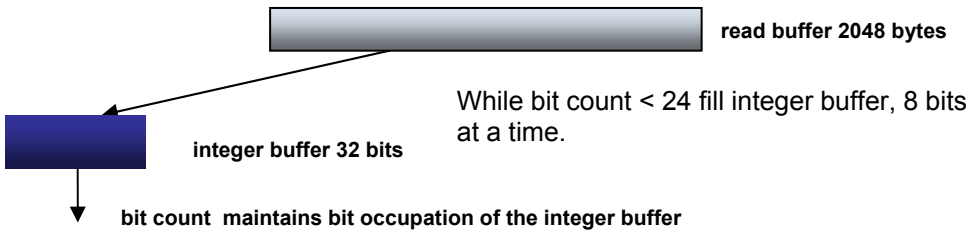
Example:

Let us look at an example of where 64-bit extended general-purpose registers can impact the performance of an MPEG2 bit stream parsing algorithm downloaded from http://www.mpeg.org

The input stream is fetched into a 2048 byte buffer called *"read buffer"*. Data is read from the read buffer into a 32-bit buffer called *"integer buffer"*. The integer buffer should contain at least 24 bits of data at any given time since that is the largest size of a bits request.

A counter called *"bit count"* maintains a running count of the *bit occupation* of the integer buffer. We will use the term bit occupation to specify the number of available bits in the integer buffer. As soon as bit count goes below 24, the integer buffer needs to be filled. Since there are at least 32-23=9 bits of free space in the integer buffer, data is read in a byte at a time. This continues until bit count exceeds or equals 24 bits.

Figure 4: MPEG2 bit stream parsing using 32-bit buffer



Listing 4: MPEG2 bit stream parsing using 32-bit buffer

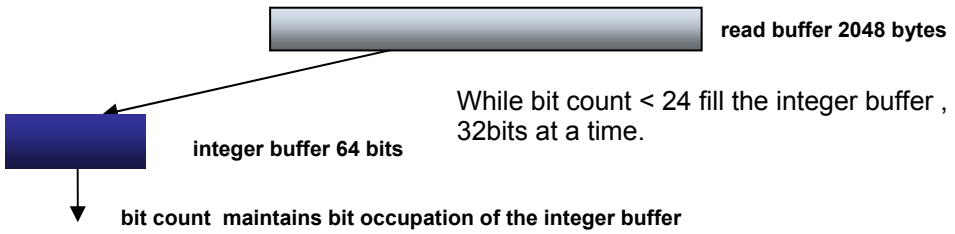
```
UINT8  RdBfr[2048];        //read buffer
UNINT  IBfr;               //integer buffer
UINT   IBfrBitCount;       //Bit Count
UINT   BufBytePos;         //Current relative read pointer into the
read buffer

UINT GetBits(UINT N)
{
       //Return the requested N bits
       UINT Val = (UINT) (IBfr >> (32 - N));

       //Adjust IBfr and IBfrBitCount after the request.
```

```
        IBfr <<= N;
        IBfrBitCount -= N;

        //If less than 24, need to refill
        while (IBfrBitCount < 24) {
               //Retrieve 8 bits at a time from RdBfr
               UINT32 New8 = * (UINT8 *) (RdBfr+ BufBytePos);

               //Adjust BufBytePos so that next 8 bits can be
               //retrieved while there is still space left in IBfr
               BufBytePos += 1;

               //Insert the 1 byte into IBfr
               //Adjust the IBfrBitCount
               IBfr |= New8 << (((32-8) - IBfrBitCount));
               IBfrBitCount += 8;
        }

        return Val;
}
```

In order to take advantage of 64-bit general purpose registers, the integer buffer is changed from a 32-bit buffer to a 64-bit buffer. As soon as bit count goes below 24 there is at least 64-23=41 bits of free space in the integer buffer. Data can now be read into the integer buffer 32 bits at a time instead of 8 bits at a time.

This will reduce the number of loads over time significantly. In addition since the bit occupation of the integer buffer will go below 24 bits less frequently, bit manipulation operations required for reading data into the integer buffer will also be reduced.

Figure 5: MPEG2 bit stream parsing using 64-bit buffer

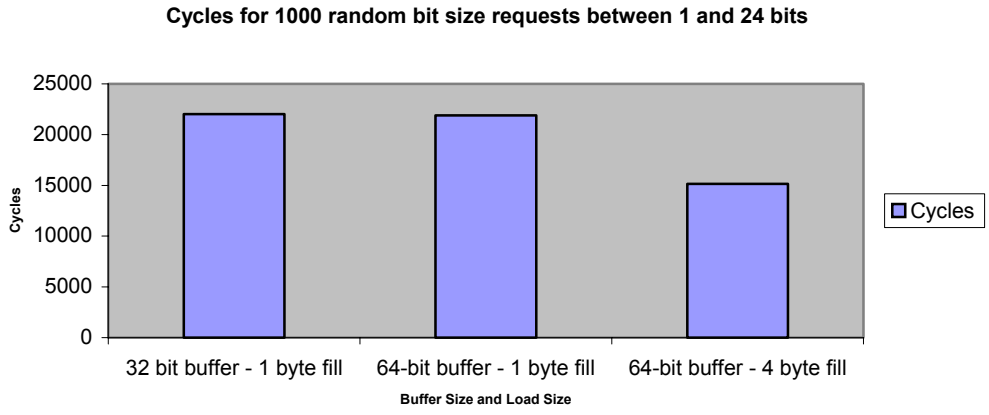


Listing 5: MPEG2 bit stream parsing using 64-bit buffer

```
UINT8  RdBfr[2048];        //read buffer
UINT64 IBfr;               //integer buffer
UINT   IBfrBitCount;       //Bit Count
UINT   BufBytePos;         //Current relative read pointer into the
                           //read buffer


UINT GetBits(UINT N)
{
        //Return the requested N bits
        UINT Val = (UINT) (IBfr >> (64 - N));

        //Adjust IBfr and IBfrBitCount after the request.
        IBfr <<= N;
```

```
        IBfrBitCount -= N;

        //If less than 24, need to refill
        while (IBfrBitCount < 24) {

                //Retrieve 32 bits at a time from RdBfr
                //The _byteswap_ulong function calls the bswap
                //instruction which converts the 32 bit integer
                //from big-endian to little-endian format.

                UINT64 New32 = _byteswap_ulong(* (UINT *)
                        (RdBfr + BufBytePos));

                //Adjust BufBytePos so that next 32 bits can be
                //retrieved while there is still space left in //IBfr
                BufBytePos += 4;

                //Insert the 4 bytes into IBfr
                //Adjust the IBfrBitCount
                IBfr |= New32 << (((64-32) - IBfrBitCount));
                IBfrBitCount += 32;
        }


        return Val;
}
```

The performance improvement between the above two routines was measured on an AMD Athlon 64 platform running 64-bit Microsoft Windows. This was done by isolating the routine and making 1000 random bits size requests from 1 bit to 24 bits. The read buffer was pre-initialized with 2048 bytes and the integer buffer was pre-initialized with 32 bits. The performance comparison is as shown below:



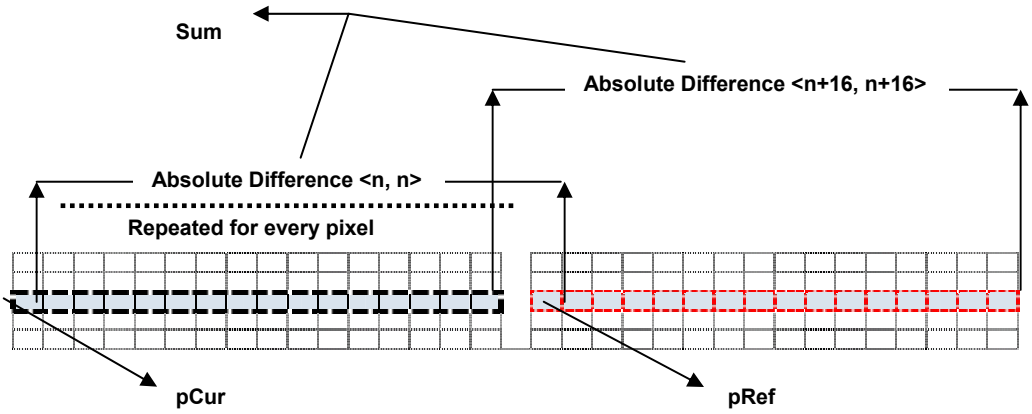
Taking Advantage Of Architectural Improvements Via Loop Unrolling

Some of the most performance critical code sequences in codecs are short loops. Since the loop body is short, such loops suffer from significant loop overhead. Unrolling such loops, even partially, can give significant performance improvement. When using intrinsics, the 64-bit Microsoft compiler expects the programmer to implement loop unrolling.

Example:

Let us look at an example in codecs, where loop unrolling can be advantageous. We will go through a sum of absolute differences (SAD) algorithm that is commonly used in motion estimation as a matching criterion between current and reference frames.

Figure 6: SAD for a row of two 16x16 blocks



Listing 6: SAD calculation for two 16x16 blocks

```
;pCur = 16-byte aligned pointer to 16x16 current block
;pRef = 1-byte aligned pointer to 16x16 Reference block
mov     esi, pCur
mov     edi, pRef
mov     ecx, iCurStride    //Stride for next row of current block
mov     edx, iRefStride    //Stride for next row of reference block
pxor    xmm7, xmm7
pxor    xmm0, xmm0
mov     eax, 16
Loop:
       movdqu        xmm2, [edi]
       psadbw        xmm2, [esi]         //Since pCur is 16-byte
aligned, we do not have to load it separately
       paddusw       xmm7, xmm2
       add           esi, ecx
       add           edi, edx
       dec           eax
       jnz           Loop
movhlps       xmm0, xmm7
paddd         xmm7, xmm0
movd          ecx, xmm7
;ecx now holds the SAD value
```

The psadbw instruction computes the absolute difference between the pair of 16 unsigned 8-bit integers. It sums the upper 8 differences and lower 8 differences. It

then packs the resulting 2 unsigned 16-bit integers into the upper and lower 64-bit elements.

The upper and the lower halves are summed appropriately after the loop. AMD Athlon 64 and AMD Opteron processors can decode and retire as many as three instructions per cycle. The loop above consists of seven instructions per iteration and executes one SAD per iteration. This gives a throughput of 3 instructions per cycle * 1 iteration per 7 instructions * 1 SAD per iteration = 3/7 SAD per cycle.

This loop is relatively small, less than 10 instructions. The number of iterations is known in advance. Hence this loop can be unrolled entirely. An unrolled implementation of the algorithm in intrinsics is listed below.

Listing 7: SAD calculation for two 16x16 blocks

```
;pCur = 16-byte aligned pointer to 16x16 current block
;pRef = Byte-aligned pointer to 16x16 Reference block

__m128i Sum, Ref, Sad;
Sum    =      _mm_xor_si128(Sum, Sum);

Ref = _mm_loadu_si128((const __m128i *) (pRef));
Sum    =_mm_sad_epu8 (Ref, *(const __m128i *)(pCur));

                    ⋮

Ref = _mm_loadu_si128((const __m128i *) (pRef + 1
*iRefStride));
Sad    =_mm_sad_epu8 (Ref, *(const __m128i *)(pCur+ 1 *
iCurStride));
_mm_add_epi32 (Sum, Sad);

                    ⋮   Repeated for next 13 rows

Ref = _mm_loadu_si128((const __m128i *) (pRef + 15
*iRefStride));
Sad    =_mm_sad_epu8 (Ref, *(const __m128i *)(pCur+ 15 *
iCurStride));
_mm_add_epi32 (Sum, Sad);

Int iSad = Sum.m128i_u32[0] + Sum.m128i_u32[2];
```

The `_mm_sad_epu8` intrinsic internally generates a `psadbw` instruction. After unrolling, the loop overhead is eliminated. Hence each row will only require 5 instructions. Since there are 16 rows, the entire unrolled loop will require 5*16 instructions. This will give a throughput of 3 instructions per cycle * 1 iteration per (5*16) instructions * 16 SADs per iteration = 3/5 SAD per cycle. This is an improvement of 40% over the rolled loop. This improvement is entirely due to the removal of loop overhead.

The added benefit of doing loop unrolling becomes evident when the developer thinks beyond the loop overhead. The 64-bit native mode has access to eight additional XMM registers compared to the 32-bit legacy mode. This allows the compiler to interleave and schedule up to sixteen independent SAD dependency chains. Thus, the full throughput capability of the execution units will be used. Having the additional registers allows the intermediate data for those parallel SAD chains to stay in the registers. The lack of extra registers in the 32-bit legacy mode would cause data to spill to and from memory. The compiler will also take advantage of scaled index addressing mode where possible. This minimizes the pointer arithmetic inside the loop. Using assembly forces the developer to

proactively use the extra registers. It also puts the onus of determining the most optimal scheduling and using other optimization techniques, baked into the compiler, on the developer. Using intrinsics delivers the additional performance free of cost to the application.

Another interesting optimization that applies to SAD algorithms is the early exit strategy. A commonly used strategy is to check the SAD value after a few iterations. If it exceeds a threshold, the routine is exited. This is repeated periodically and a progressively lower threshold is used each time.

Developers should be careful not use such an excessive exit strategy. Checking the SAD value in the SIMD algorithm requires the data to be transferred via memory from an XMM register to a general purpose register. Then there is the overhead of the conditional jump, which can be expensive, if mispredicted. Such checks also convert the algorithm into one long dependency chain, preventing the full throughput of the execution units.

A recommended strategy would be to test the SAD value just once early on in the loop. If the early exit is not taken, allowing the rest of the SIMD SAD calculation to continue without any checks will give the most optimal performance. On some popular codecs, a gain of up to 6% has been observed by using this conservative exit strategy.

Using Aligned Memory Accesses

An SSE/SSE2 algorithm often requires loading and storing data 16 bytes at a time to match the size of XMM registers to be operated upon. If data is 16-byte aligned, the developer can take advantage of the aligned load and store intrinsics which will use the aligned load and store instructions. If the data is not 16-byte aligned then the developer must use the unaligned load and store intrinsics, which will generate the unaligned load and store instructions.

Modern x86 processors use complex instructions that are internally broken down into a sequence of simpler operations called micro-ops. The AMD Athlon 64 and AMD Opteron processors support two types of instructions called fast-path and vector-path. Fast-path instructions are decoded into one or two micro-ops. Vector-path instructions are decoded into three or more micro-ops. While more than one fast-path instruction can be dispatched in a cycle, only one vector-path instruction can be dispatched in a cycle. Vector-path instructions also have higher static-execution latencies than fast-path instructions.

All the aligned load and store instructions are fast-path instructions. While all unaligned loads and stores are vector-path instructions.

Below is a listing of various aligned and unaligned load and store intrinsics and the instructions generated by them. Also provided is the decode-type and static execution latency in clock cycles of the instruction.

SSE2 Intrinsics	SSE2 Instructions	Decode Type	Static execution latency
_mm_load_si128 _mm_store_si128	movdqa	Fast-path double	Load – 2 Store – 3
_mm_load_ps _mm_store_ps	movaps	Fast-path double	Load – 4 Store – 3
_mm_load_pd _mm_store_pd	movapd	Fast-path double	Load – 2 Store – 3

_mm_loadu_si128 _mm_storeu_si128	movdqu	Vector Path	Load - 7 Store - 4
_mm_loadu_ps _mm_storeu_ps	movups	Vector Path	Load - 7 Store - 4
_mm_loadu_pd _mm_storeu_pd	movupd	Vector Path	Load - 7 Store - 4

Using 16-byte alignment allows the developer to use aligned load and store intrinsics. It can significantly speed up the loads and stores and should be preferred. Using 16-byte alignment also avoids cache-line splitting loads, which could significantly reduce cache performance.

The Microsoft Windows compiler for AMD64 platforms exposes the `__declspec(align(#))` directive to allow the programmer precise control over the alignment of user-defined data. Let us look at an example of where aligned loads and stores can be beneficial.

Example:

Consider Discrete Cosine Transform (DCT), a time-consuming step during video compression. DCT is used to transform a block of pixel values into a set of "spatial frequency" coefficients. A 2-dimensional (2D) DCT can be implemented in two steps. First a 1-dimensional (1D) DCT is performed on each row of a block of pixels. The next step performs another 1D DCT on each column of the block of pixels generated by the first step.

It is beneficial to align the intermediate matrix produced by the first 1D DCT on a 16-byte boundary. This allows the stores produced by the first 1D DCT to be aligned stores. It also allows the loads produced by the second 1D DCT to be aligned loads. The intermediate matrix should be aligned using the `__declspec(align(#))` directive, instead of explicit alignment with bit manipulation operations as shown below.

```
char d_unaligned[64];
char *d;
d = (char *)(( (size_t)d_unaligned +
15)&~15);
```

→

```
__declspec(align(16)) char d[64];
```

Aligning the memory accesses also allows the developer to take advantage of the store to load forwarding (STLF) mechanism of the AMD Athlon 64 and AMD Opteron processors. STLF refers to the process of a load reading data from the load-store (LS) buffer, even before the store has written data to the data cache.

If either the load or the store data is misaligned then STLF can not occur. In these cases, the load cannot complete until the store has written to the data cache and thus retired. A store can not retire until all instructions in the reorder buffer, up to and including the store complete and retire out of the reorder buffer. Effectively, the load has a false dependency on every instruction up to the store.

Due to the significant depth of the LS buffer, any load that is dependent on a store that cannot bypass data through the LS buffer may experience significant delays. This delay can be up to tens of clock cycles, where the exact delay is a function of pipeline conditions.

In general, the user should align all local intermediate buffers to 16 byte boundaries in order to take advantage of STLF and reduce load and store latencies on AMD64 platforms.

Reducing The Impact Of Misaligned Memory Accesses

Most codec software suffers from inherent misalignment, given that the macro-blocks that are operated upon can start at any pixel in a frame. When 16-byte alignment is not possible, developers should avail themselves of certain optimization techniques to reduce the impact of misalignment.

Load high and low parts of XMM registers separately

Instead of using unaligned load and store instructions, developers should use the SSE `movlps` and `movhps` instructions for integer data and single precision data.

SSE Intrinsics	SSE Instructions	Decode Type	Static execution latency
_mm_loadl_pi	movlps	Fast-path single	Load - 2 Store - 2
_mm_loadh_pi	movhps	Fast-path single	Load - 2 Store - 2

In general, loads and stores are not sensitive to data types. We will not discuss the impact of mixing data types on instructions other than loads and stores in this paper. Additional information on this topic can be obtained in the Software Optimization guide for AMD Athlon 64 and AMD Opteron processors.

Example:

A simple utility routine, shown below, should allow the user to load two unaligned groups of eight pixels each into an XMM register.

Listing 8: Utility routine for unaligned load

```
#define Pixelb char
__forceinline __m128i Pack_2Q (const Pixelb *p1, const Pixelb *p2){
    __m128i m;
    m = _mm_loadl_pi (m, (__m64 *) p1);
    m = _mm_loadh_pi (m, (__m64 *) p2);
    return (m);
}
```

Eliminate redundant loads

If data cannot be aligned, the developer should look for opportunities to avoid redundant loads. This should be applied to both aligned and misaligned loads where possible.

Example:

Without getting too engrossed in the theory of motion estimation, let's go through an algorithm for vertical half-pel interpolation, where an intermediate pixel in the output block is a weighted combination of the corresponding pixel in the input block and its nearest vertical neighbor below.

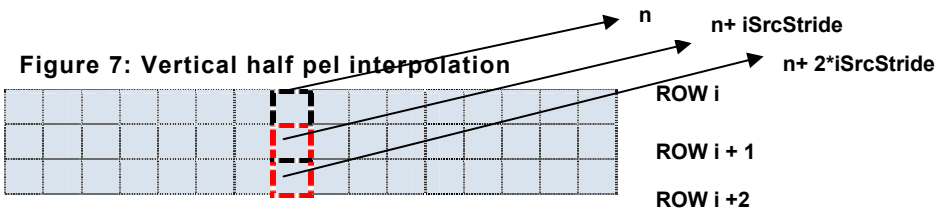


Figure 7: Vertical half pel interpolation

Pixels n and n+ iSrcStride are averaged during the first iteration in the input image to generate the intermediate pixel that will be inserted between them in the output image. This calculation is repeated for every pixel in a 16x16 block.

Let us consider three rows (i, i + 1 and i + 2) being processed at a point in time. Using an SSE2 algorithm, all 16 pixels in row i can be averaged with 16 pixels in row i+1 at a time. Likewise, all 16 pixels in row i+1 can be averaged will 16 pixels in row i+2 at a time.

A simple implementation of this routine in SSE2 assembly is shown below:

Listing 9: Vertical half pel interpolation with redundant unaligned loads

```
;pSrc = unaligned pointer to 16x16 block of 1 byte pixels
;pDst = unaligned pointer to 16x16 block of 1 byte pixels
mov  esi, pSrc
mov  edi, pDst
mov  ecx, iSrcStride
mov  edx, iDstStride
mov  eax, 16
Loop:
       movdqu        xmm1, [esi]
       movdqu        xmm2, [esi + ecx]
       pavgb         xmm1, xmm2
       movdqu        [edi], xmm1
       add           esi, ecx
       add           edi, edx
       dec           eax
       jnz           Loop
```

Since the entire row i+1 is common between the two passes, it only needs to be loaded once. This reduces the overall number of unaligned loads by 50%. The implementation of this algorithm in SSE2 assembly is as shown below:

Listing 10: Preferred Vertical half pel interpolation with 50% fewer unaligned loads

```
;pSrc = unaligned pointer to 16x16 block of 1 byte pixels
;pDst = unaligned pointer to 16x16 block of 1 byte pixels
mov    esi, pSrc
mov    edi, pDst
mov    ecx, iSrcStride
mov    edx, iDstStride
movdqu xmm1, [esi]
mov  eax, 16
Loop:
       movdqu        xmm2, [esi + ecx]
       pavgb         xmm1, xmm2
       movdqu        [edi], xmm1
       add           esi, ecx
       add           edi, edx
       movdqa        xmm1, xmm2
       dec           eax
       jnz           Loop
```

This preferred assembly listing will be generated by the intrinsics shown below:

Listing 11: Preferred Vertical half pel interpolation in Intrinsics

```
;pSrc = unaligned pointer to 16x16 block of 1 byte pixels
;pDst = unaligned pointer to 16x16 block of 1 byte pixels
__m128i Cur_Row, Next_Row;
Cur_Row = _mm_loadu_si128((__m128i *)pSrc);
for (int i=1; i<= 16; i++)
{
       Next_Row = _mm_loadu_si128((__m128i *)(pSrc + i * iSrcStride);
       Cur_Row = _mm_avg_epu8(Cur_Row, Next_Row);
       _mm_storeu_si128((__m128i *)(pDst), Cur_Row);
       pSrc   += iSrcStride;
       pDst   += iDstStride;
       Cur_Row = Next_Row;
}
```

Software Prefetching

Prefetching refers to the technique of hiding memory latencies and getting the data for operations beforehand. Codecs either process the entire image or process macro-blocks of data. When the entire image is being processed row by row contiguously, the hardware prefetcher on AMD64 platforms will do an efficient job. The hardware prefetcher brings in 64 bytes (cache line) at a time. When the hardware prefetcher detects an access to cache line L, followed by cache line L+1, it will initiate a prefetch to the subsequent cache line.

The hardware prefetcher has some limitations. It cannot cross page boundaries. It only fills the L2 cache. It may not detect access patterns with large strides. All these limitations can be overcome with the correct use of software prefetches.

Software prefetches can also be given hints about different kinds of data. For data that is read and will be needed again soon, developers should use the “Read” prefetches. For data that is discarded after the first use, developers should use the “Non-temporal read” version of prefetch. For data, which will soon be written to, developers should use the “Write” prefetches.

Below is a listing of the various intrinsics that can generate the corresponding prefetch instructions.

SSE2 Intrinsics	SSE2 Instructions	Type of prefetch	Cache filled and Cache evicted to.
_mm_prefetch(void* address,_MM_HINT_T0)	prefetch	Read prefetch	Fills L1 data cache. When evicted, data sent to L2 data cache.
_mm_prefetch(void* address,_MM_HINT_NTA)	prefetchnta	Non-temporal read prefetch	Fills one way of L1 data cache. When evicted, data sent directly to memory if modified and not to L2 data cache.
_m_prefetchw(void* address)	prefetchw	Write prefetch.	Fills L1 data cache. When evicted, data sent to L2 data cache.

As an example, let us consider doing the color space conversion of the entire image. The color converted output image can be prefetched with a write prefetch as it will be written to soon. The developer might even consider using streaming stores

in place of write prefetches, if the output image is not needed again soon after the write. Streaming stores or write combining instructions write data directly to the memory bypassing the cache. The intrinsics that can be used for streaming stores are listed below.

SSE/SSE2 Intrinsics	SSE/SSE2 Instructions	Type of store
_mm_stream_si128(m128i* address, _m128i var)	movntdq	Non-temporal write
_mm_stream_ps(void* address, _m128i var)	movntps	
_mm_stream_pd(void* address, _m128i var)	movntpd	

However, there are some pitfalls that the programmer should be mindful of when using software prefetches.

Using software prefetches requires the developer to specify the address to start prefetching from. When determining how far ahead to prefetch, the basic guideline is to initiate the prefetch far enough ahead to give it time to complete. To determine the optimal prefetch distance, the user should use empirical benchmarking on the target platform when possible. Prefetching three or four cache lines or loop iterations ahead is a good starting point. Trying to prefetch too far ahead impairs performance.

The optimal prefetch distance is platform specific. For applications with long life cycles, developers might try to tune the optimal prefetch distance in the field.

The developer should also avoid excessive and redundant prefetching that can lead to performance penalties since it may block the decode and dispatch of other instructions.

While the prefetch is accessing the memory, the processor should be kept busy with computations to overlap that time. As a general rule, software prefetches should not be used when processing 8x8 blocks, since there are usually not enough computations to overlap the latency of a prefetch. Prefetches could be useful when processing 16x16 or bigger blocks, where sufficient computation is possible. But such tuning is application specific and some experimentation will be necessary.

Porting And Performance Evaluation Tools

High performance codecs have traditionally been developed in two stages. The first stage is to develop the reference C/C++ code. The next stage is to determine and analyze the performance critical parts of the codec and take advantage of SIMD algorithms to speed them up. Most of the 32-bit codecs have these performance critical sections coded in MMX or SSE/SSE2 inline assembly.

The first step in migrating codecs to AMD64 platforms is porting the reference C/C++ code to build correctly on the AMD64 platform. Developers can use existing Microsoft Visual Studio version 6, .Net or even the next generation development environment called Whidbey. For additional information on how to configure and use these environments for AMD64 platforms, developers should refer to Winhec 2004 white paper on "Configuring Microsoft Visual Studio Projects to support the AMD64 architecture".

The next step is the conversion and performace tuning of the critical modules. Before embarking on this step, it is useful to determine the hot spots in the codec. Converting the top few hot spots usually gives the most performance improvement. AMD's Code Analyst profiling tool can be used to determine these hot spots and dig into the generated code. For additional information on Code Analyst and how to use it, refer to the DevX article CodeAnalyst: Getting in touch with your inner code. Developers can download CodeAnalyst from the AMD developer Program site.

Developers can also find a perl script that can do a partial conversion of MMX assembly code to Intrinsics at the at the AMD64 developer resource kit site. This script can only perform a dumb conversion from assembly to Intrinsics, but can be expanded to add more cases. The onus of using the optimization techniques stated above and handling corner cases still rests with the developer.

About the Author

Harsha Jagasia works at Advanced Micro Devices in Austin. Her main expertise lies in the area of optimizing compilers, libraries and applications for AMD Athlon 64 and AMD Opteron processors.

Acknowledgements

The author would like to thank these folks at AMD for their support, ideas and review while writing this paper: Frank Gorishek, Tom Deneau, Evandro Menezes, Chip Freitag, Dan McCabe, Wei-Lien Hsu, Jim Conyngham, Mark Santaniello and Kent Knox.

Resources and References

Configuring Microsoft Visual Studio Projects to support the AMD64 architecture, Winhec 2004 paper by Evandro Menezes, Advanced Micro Devices.

Porting and Optimizing Applications on 64-bit Windows for AMD64 Architecture, Winhec 2004 paper by Mike Wall, Advanced Micro Devices.

Software Optimization Guide for AMD Athlon™ 64 and AMD Opteron™ Processors, Advanced Micro Devices.

AMD64 developer resource kit site, Advanced Micro Devices.

AMD developer Program, Advanced Micro Devices

CodeAnalyst: Getting in touch with your inner code, DevX article by Alan Zeichick, Camden Associates

http://www.mpeg.org, MPEG resources

Porting and optimizing applications for AMD64 architecture on Microsoft Windows, presentation by Mike Wall, Advanced Micro Devices.